$$\boxed{\Psi \vdash e : A \rightsquigarrow s}$$

$$\frac{x : A \in \Psi}{\Psi \vdash x : A \rightsquigarrow x}\ \textsc{Var} \qquad \frac{}{\Psi \vdash n : \mathsf{Int} \rightsquigarrow n}\ \textsc{Nat} \qquad \frac{\Psi, a \vdash e : A \rightsquigarrow s}{\Psi \vdash e : \forall a.A \rightsquigarrow \Lambda a.s}\ \textsc{Gen}$$

$$\frac{\Psi, x : A \vdash e : B \rightsquigarrow s}{\Psi \vdash \lambda x : A.\ e : A \to B \rightsquigarrow \lambda x : A.\ s}\ \textsc{LamAnn} \qquad \frac{\Psi, x : \tau \vdash e : B \rightsquigarrow s}{\Psi \vdash \lambda x.\ e : \tau \to B \rightsquigarrow \lambda x : \tau.\ s}\ \textsc{Lam}$$

$$\frac{\Psi \vdash e_1 : A \rightsquigarrow s_1 \qquad \Psi \vdash A \triangleright A_1 \to A_2 \qquad \Psi \vdash e_2 : A_3 \rightsquigarrow s_2 \qquad \Psi \vdash A_3 \lesssim A_1}{\Psi \vdash e_1\ e_2 : A_2 \rightsquigarrow (\langle A \hookrightarrow A_1 \to A_2 \rangle\ s_1)\ (\langle A_3 \hookrightarrow A_1 \rangle\ s_2)}\ \textsc{App}$$

$$\boxed{\Psi \vdash A \triangleright A_1 \to A_2}$$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] \triangleright A_1 \to A_2}{\Psi \vdash \forall a.A \triangleright A_1 \to A_2}\ \textsc{M-Forall}$$

$$\frac{}{\Psi \vdash (A_1 \to A_2) \triangleright (A_1 \to A_2)}\ \textsc{M-Arr} \qquad \frac{}{\Psi \vdash \star \triangleright \star \to \star}\ \textsc{M-Unknown}$$

Fig. 8. Declarative typing

4 Gradually Typed Implicit Polymorphism

In Sect. 3 we introduced the consistent subtyping relation that accommodates polymorphic types. In this section we continue with the development by giving a declarative type system for predicative implicit polymorphism that employs the consistent subtyping relation. The declarative system itself is already quite interesting as it is equipped with both higher-rank polymorphism and the unknown type. The syntax of expressions in the declarative system is given below:

$$\text{Expressions} \quad e ::= x \mid n \mid \lambda x : A.\ e \mid \lambda x.\ e \mid e\ e$$

4.1 Typing in Detail

Figure 8 gives the typing rules for our declarative system (the reader is advised to ignore the gray-shaded parts for now). Rule Var extracts the type of the variable from the typing context. Rule Nat always infers integer types. Rule LamAnn puts x with type annotation A into the context, and continues type checking the body e. Rule Lam assigns a monotype τ to x, and continues type checking the body e. Gradual types and polymorphic types are introduced via annotations explicitly. Rule Gen puts a fresh type variable a into the type context and generalizes the typing result A to $\forall a.A$. Rule App first infers the type of e_1, then the matching judgment $\Psi \vdash A \triangleright A_1 \to A_2$ extracts the domain type A_1 and the codomain type A_2 from type A. The type A_3 of the argument e_2 is then compared with A_1 using the consistent subtyping judgment.

Matching. The matching judgment of Siek et al. [25] can be extended to polymorphic types naturally, resulting in $\Psi \vdash A \triangleright A_1 \to A_2$. In M-Forall, a monotype τ is guessed to instantiate the universal quantifier a. This rule is inspired by the *application judgment* $\Phi \vdash A \bullet e \Rightarrow C$ [11], which says that if we apply a term of type A to an argument e, we get something of type C. If A is a polymorphic type, the judgment works by guessing instantiations until it reaches an arrow type. Matching further simplifies the application judgment, since it is independent of typing. Rule M-Arr and M-Unknown are the same as Siek et al. [25]. M-Arr returns the domain type A_1 and range type A_2 as expected. If the input is $\star$, then M-Unknown returns $\star$ as both the type for the domain and the range.

Note that matching saves us from having a subsumption rule (Sub in Fig. 2). the subsumption rule is incompatible with consistent subtyping, since the latter is not transitive. A discussion of a subsumption rule based on normal subtyping can be found in the appendix.

4.2 Type-Directed Translation

We give the dynamic semantics of our language by translating it to $\lambda\mathsf{B}$. Below we show a subset of the terms in $\lambda\mathsf{B}$ that are used in the translation:

$$\text{Terms} \quad s ::= x \mid n \mid \lambda x : A.\ s \mid \Lambda a.s \mid s_1\ s_2 \mid \langle A \hookrightarrow B \rangle\ s$$

A cast $\langle A \hookrightarrow B \rangle\ s$ converts the value of term s from type A to type B. A cast from A to B is permitted only if the types are *compatible*, written $A \prec B$, as briefly mentioned in Sect. 3.1. The syntax of types in $\lambda\mathsf{B}$ is the same as ours.

The translation is given in the gray-shaded parts in Fig. 8. The only interesting case here is to insert explicit casts in the application rule. Note that there is no need to translate matching or consistent subtyping, instead we insert the source and target types of a cast directly in the translated expressions, thanks to the following two lemmas:

Lemma 1 ($\triangleright$ to $\prec$). *If* $\Psi \vdash A \triangleright A_1 \to A_2$, *then* $A \prec A_1 \to A_2$.

Lemma 2 ($\lesssim$ to $\prec$). *If* $\Psi \vdash A \lesssim B$, *then* $A \prec B$.

In order to show the correctness of the translation, we prove that our translation always produces well-typed expressions in $\lambda\mathsf{B}$. By Lemmas 1 and 2, we have the following theorem:

Theorem 2 (Type Safety). *If* $\Psi \vdash e : A \rightsquigarrow s$, *then* $\Psi \vdash^B s : A$.

Parametricity. An important semantic property of polymorphic types is *relational parametricity* [19]. The parametricity property says that all instances of a polymorphic function should behave *uniformly*. A classic example is a function with the type $\forall a.a \to a$. The parametricity property guarantees that a value of this type must be either the identity function (i.e., $\lambda x.x$) or the undefined function (one which never returns a value). However, with the addition of the unknown type $\star$, careful measures are to be taken to ensure parametricity. This is exactly the circumstance that $\lambda\mathsf{B}$ was designed to address. Ahmed et al. [2] proved that $\lambda\mathsf{B}$ satisfies relational parametricity. Based on their result, and by Theorem 2, parametricity is preserved in our system.

Ambiguity from Casts. The translation does not always produce a unique target expression. This is because when we guess a monotype τ in rule M-ForAll and CS-ForAllL, we could have different choices, which inevitably leads to different types. Unlike (non-gradual) polymorphic type systems [11,18], the choice of monotypes could affect runtime behaviour of the translated programs, since they could appear inside the explicit casts. For example, the following shows two possible translations for the same source expression $\lambda x : \star.\ f\ x$, where the type of f is instantiated to $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Bool}$, respectively:

$$
\begin{aligned}
f : \forall a.a \to a \vdash (\lambda x : \star.\ f\ x) : \star \to \mathsf{Int} \\
&\rightsquigarrow (\lambda x : \star.\ (\langle \forall a.a \to a \hookrightarrow \mathsf{Int} \to \mathsf{Int} \rangle\ f)\ (\langle \star \hookrightarrow \mathsf{Int} \rangle\ x)) \\
f : \forall a.a \to a \vdash (\lambda x : \star.\ f\ x) : \star \to \mathsf{Bool} \\
&\rightsquigarrow (\lambda x : \star.\ (\langle \forall a.a \to a \hookrightarrow \mathsf{Bool} \to \mathsf{Bool} \rangle\ f)\ (\langle \star \hookrightarrow \mathsf{Bool} \rangle\ x))
\end{aligned}
$$

If we apply $\lambda x : \star.\ f\ x$ to 3, which is fine since the function can take any input, the first translation runs smoothly in $\lambda\mathsf{B}$, while the second one will raise a cast error (Int cannot be cast to Bool). Similarly, if we apply it to true, then the second succeeds while the first fails. The culprit lies in the highlighted parts where any instantiation of a would be put inside the explicit cast. More generally, any choice introduces an explicit cast to that type in the translation, which causes a runtime cast error if the function is applied to a value whose type does not match the guessed type. Note that this does not compromise the type safety of the translated expressions, since cast errors are part of the type safety guarantees.

Coherence. The ambiguity of translation seems to imply that the declarative system is *incoherent*. A semantics is coherent if distinct typing derivations of the same typing judgment possess the same meaning [20]. We argue that the declarative system is "coherent up to cast errors" in the sense that a well-typed program produces a unique value, or results in a cast error. In the above example, whatever the translation might be, applying $\lambda x : \star.\ f\ x$ to 3 either results in a cast error, or produces 3, nothing else.

This discrepancy is due to the guessing nature of the *declarative* system. As far as the declarative system is concerned, both $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Bool}$ are equally acceptable. But this is not the case at runtime. The acute reader may have found that the *only* appropriate choice is to instantiate f to $\star \to \star$. However, as specified by rule M-ForAll in Fig. 8, we can only instantiate type variables to monotypes, but $\star$ is *not* a monotype! We will get back to this issue in Sect. 6.2 after we present the corresponding algorithmic system in Sect. 5.

4.3 Correctness Criteria

Siek et al. [25] present a set of properties that a well-designed gradual typing calculus must have, which they call the refined criteria. Among all the criteria, those related to the static aspects of gradual typing are well summarized

by Cimini and Siek [8]. Here we review those criteria and adapt them to our notation. We have proved in Coq that our type system satisfies all these criteria.

Lemma 3 (Correctness Criteria)

- ***Conservative extension:*** *for all static* Ψ, e, *and* A,
  - *if* $\Psi \vdash^{OL} e : A$, *then there exists* B, *such that* $\Psi \vdash e : B$, *and* $\Psi \vdash B <: A$.
  - *if* $\Psi \vdash e : A$, *then* $\Psi \vdash^{OL} e : A$
- ***Monotonicity w.r.t. precision:*** *for all* Ψ, e, e', A, *if* $\Psi \vdash e : A$, *and* $e' \sqsubseteq e$, *then* $\Psi \vdash e' : B$, *and* $B \sqsubseteq A$ *for some* B.
- ***Type Preservation of cast insertion:*** *for all* Ψ, e, A, *if* $\Psi \vdash e : A$, *then* $\Psi \vdash e : A \rightsquigarrow s$, *and* $\Psi \vdash^{B} s : A$ *for some* s.
- ***Monotonicity of cast insertion:*** *for all* $\Psi, e_1, e_2, e_1', e_2', A$, *if* $\Psi \vdash e_1 : A \rightsquigarrow e_1'$, *and* $\Psi \vdash e_2 : A \rightsquigarrow e_2'$, *and* $e_1 \sqsubseteq e_2$, *then* $\Psi \shortmid \Psi \vdash e_1' \sqsubseteq^{B} e_2'$.

The first criterion states that the gradual type system should be a conservative extension of the original system. In other words, a *static* program that is typeable in the Odersky-Läufer type system if and only if it is typeable in the gradual type system. A static program is one that does not contain any type $\star$[7]. However since our gradual type system does not have the subsumption rule, it produces more general types.

The second criterion states that if a typeable expression loses some type information, it remains typeable. This criterion depends on the definition of the precision relation, written $A \sqsubseteq B$, which is given in the appendix. The relation intuitively captures a notion of types containing more or less unknown types ($\star$). The precision relation over types lifts to programs, i.e., $e_1 \sqsubseteq e_2$ means that e_1 and e_2 are the same program except that e_2 has more unknown types.

The first two criteria are fundamental to gradual typing. They explain for example why these two programs $(\lambda x : \mathsf{Int}.\ x + 1)$ and $(\lambda x : \star.\ x + 1)$ are typeable, as the former is typeable in the Odersky-Läufer type system and the latter is a less-precise version of it.

The last two criteria relate the compilation to the cast calculus. The third criterion is essentially the same as Theorem 2, given that a target expression should always exist, which can be easily seen from Fig. 8. The last criterion ensures that the translation must be monotonic over the precision relation $\sqsubseteq$.

As for the dynamic guarantee, things become a bit murky for two reasons: (1) as we discussed before, our declarative system is incoherent in that the runtime behaviour of the same source program can vary depending on the particular translation; (2) it is still unknown whether dynamic guarantee holds in $\lambda\mathsf{B}$. We will have more discussion on the dynamic guarantee in Sect. 6.3.

5 Algorithmic Type System

In this section we give a bidirectional account of the algorithmic type system that implements the declarative specification. The algorithm is largely inspired by the

[7] Note that the term *static* has appeared several times with different meanings.

Expressions	$e ::= x \mid n \mid \lambda x : A.\ e \mid \lambda x.\ e \mid e\ e \mid e : A$
Types	$A, B ::= \mathsf{Int} \mid a \mid \widehat{a} \mid A \to B \mid \forall a.A \mid \star$
Monotypes	$\tau, \sigma ::= \mathsf{Int} \mid a \mid \widehat{a} \mid \tau \to \sigma$
Contexts	$\Gamma, \Delta, \Theta ::= \varnothing \mid \Gamma, x : A \mid \Gamma, a \mid \Gamma, \widehat{a} \mid \Gamma, \widehat{a} = \tau$
Complete Contexts	$\Omega ::= \varnothing \mid \Omega, x : A \mid \Omega, a \mid \Omega, \widehat{a} = \tau$

Fig. 9. Syntax of the algorithmic system

$$\boxed{\Gamma \vdash A \lesssim B \dashv \Delta}$$

$$\frac{}{\Gamma[a] \vdash a \lesssim a \dashv \Gamma[a]}\ \text{ACS-TVar} \qquad \frac{}{\Gamma[\widehat{a}] \vdash \widehat{a} \lesssim \widehat{a} \dashv \Gamma[\widehat{a}]}\ \text{ACS-ExVar}$$

$$\frac{}{\Gamma \vdash \mathsf{Int} \lesssim \mathsf{Int} \dashv \Gamma}\ \text{ACS-Int} \qquad \frac{}{\Gamma \vdash \star \lesssim A \dashv \Gamma}\ \text{ACS-UnknownL} \qquad \frac{}{\Gamma \vdash A \lesssim \star \dashv \Gamma}\ \text{ACS-UnknownR}$$

$$\frac{\Gamma \vdash B_1 \lesssim A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 \lesssim [\Theta]B_2 \dashv \Delta}{\Gamma \vdash A_1 \to A_2 \lesssim B_1 \to B_2 \dashv \Delta}\ \text{ACS-Fun}$$

$$\frac{\Gamma, a \vdash A \lesssim B \dashv \Delta, a, \Theta}{\Gamma \vdash A \lesssim \forall a.B \dashv \Delta}\ \text{ACS-ForallR} \qquad \frac{\Gamma, \widehat{a} \vdash A[a \mapsto \widehat{a}] \lesssim B \dashv \Delta}{\Gamma \vdash \forall a.A \lesssim B \dashv \Delta}\ \text{ACS-ForallL}$$

$$\frac{\widehat{a} \notin fv(A) \qquad \Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox A \dashv \Delta}{\Gamma[\widehat{a}] \vdash \widehat{a} . \lesssim A \dashv \Delta}\ \text{ACS-InstL} \qquad \frac{\widehat{a} \notin fv(A) \qquad \Gamma[\widehat{a}] \vdash A \lessapprox \widehat{a} \dashv \Delta}{\Gamma[\widehat{a}] \vdash A \lesssim \widehat{a} \dashv \Delta}\ \text{ACS-InstR}$$

Fig. 10. Algorithmic consistent subtyping

algorithmic bidirectional system of Dunfield and Krishnaswami [11] (henceforth DK system). However our algorithmic system differs from theirs in three aspects: (1) the addition of the unknown type $\star$; (2) the use of the matching judgment; and (3) the approach of *gradual inference only producing static types* [12]. We then prove that our algorithm is both sound and complete with respect to the declarative type system. Full proofs can be found in the appendix.

Algorithmic Contexts. The algorithmic context Γ is an *ordered* list containing declarations of type variables a and term variables $x : A$. Unlike declarative contexts, algorithmic contexts also contain declarations of existential type variables $\widehat{a}$, which can be either unsolved (written $\widehat{a}$) or solved to some monotype (written $\widehat{a} = \tau$). Complete contexts Ω are those that contain no unsolved existential type variables. Figure 9 shows the syntax of the algorithmic system. Apart from expressions in the declarative system, we have annotated expressions $e : A$.

5.1 Algorithmic Consistent Subtyping and Instantiation

Figure 10 shows the algorithmic consistent subtyping rules. The first five rules do not manipulate contexts. Rule ACS-Fun is a natural extension of its declarative counterpart. The output context of the first premise is used by the second

$$\boxed{\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta}$$

$$\frac{\Gamma \vdash \tau}{\Gamma, \widehat{a}, \Gamma' \vdash \widehat{a} \lessapprox \tau \dashv \Gamma, \widehat{a} = \tau, \Gamma'}\ \textsc{InstLSolve} \qquad \frac{}{\Gamma[\widehat{a}][\widehat{b}] \vdash \widehat{a} \lessapprox \widehat{b} \dashv \Gamma[\widehat{a}][\widehat{b} = \widehat{a}]}\ \textsc{InstLReach}$$

$$\frac{}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox \star \dashv \Gamma[\widehat{a}]}\ \textsc{InstLSolveU} \qquad \frac{\Gamma[\widehat{a}], b \vdash \widehat{a} \lessapprox B \dashv \Delta, b, \Delta'}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox \forall b.B \dashv \Delta}\ \textsc{InstLAllR}$$

$$\frac{\Gamma[\widehat{a}_2, \widehat{a}_1, \widehat{a} = \widehat{a}_1 \to \widehat{a}_2] \vdash A_1 \lessapprox \widehat{a}_1 \dashv \Theta \qquad \Theta \vdash \widehat{a}_2 \lessapprox [\Theta]A_2 \dashv \Delta}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox A_1 \to A_2 \dashv \Delta}\ \textsc{InstLArr}$$

Fig. 11. Algorithmic instantiation

premise, and the output context of the second premise is the output context of the conclusion. Note that we do not simply check $A_2 \lesssim B_2$, but apply Θ to both types (e.g., $[\Theta]A_2$). This is to maintain an important invariant that types are fully applied under input context Γ (they contain no existential variables already solved in Γ). The same invariant applies to every algorithmic judgment. Rule ACS-ForallR looks similar to its declarative counterpart, except that we need to drop the trailing context a, Θ from the concluding output context since they become out of scope. Rule ACS-ForallL generates a fresh existential variable $\widehat{a}$, and replaces a with $\widehat{a}$ in the body A. The new existential variable $\widehat{a}$ is then added to the premise's input context. As a side note, when both types are quantifiers, then either ACS-ForallR or ACS-ForallR could be tried. In practice, one can apply ACS-ForallR eagerly. The last two rules together check consistent subtyping with an unsolved existential variable on one side and an arbitrary type on the other side by the help of the instantiation judgment.

The judgment $\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta$ defined in Fig. 11 instantiates unsolved existential variables. Judgment $\widehat{a} \lessapprox A$ reads "instantiate $\widehat{a}$ to a consistent subtype of A". For space reasons, we omit its symmetric judgement $\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta$. Rule InstLSolve and rule InstLReach set $\widehat{a}$ to τ and $\widehat{b}$ in the output context, respectively. Rule InstLSolveU is similar to ACS-UnknownR in that we put no constraint on $\widehat{a}$ when it meets the unknown type $\star$. This design decision reflects the point that type inference only produces static types [12]. We will get back to this point in Sect. 6.2. Rule InstLAllR is the instantiation version of rule ACS-ForallR. The last rule InstLArr applies when $\widehat{a}$ meets a function type. It follows that the solution must also be a function type. That is why, in the first premise, we generate two fresh existential variables $\widehat{a}_1$ and $\widehat{a}_2$, and insert them just before $\widehat{a}$ in the input context, so that the solution of $\widehat{a}$ can mention them. Note that $A_1 \lessapprox \widehat{a}_1$ switches to the other instantiation judgment.

5.2 Algorithmic Typing

We now turn to the algorithmic typing rules in Fig. 12. The algorithmic system uses bidirectional type checking to accommodate polymorphism. Most of

$$\boxed{\Gamma \vdash e \Rightarrow A \dashv \Delta}$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \dashv \Gamma}\ \textsc{AVar} \qquad \frac{}{\Gamma \vdash n \Rightarrow \mathsf{Int} \dashv \Gamma}\ \textsc{ANat}$$

$$\frac{\Gamma, \widehat{a}, \widehat{b}, x : \widehat{a} \vdash e \Leftarrow \widehat{b} \dashv \Delta, x : \widehat{a}, \Theta}{\Gamma \vdash \lambda x.\ e \Rightarrow \widehat{a} \to \widehat{b} \dashv \Delta}\ \textsc{ALamU} \qquad \frac{\Gamma, x : A \vdash e \Rightarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x : A.\ e \Rightarrow A \to B \dashv \Delta}\ \textsc{ALamAnnA}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash e \Leftarrow A \dashv \Delta}{\Gamma \vdash e : A \Rightarrow A \dashv \Delta}\ \textsc{AAnno}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta_1 \qquad \Theta_1 \vdash [\Theta_1]A \triangleright A_1 \to A_2 \dashv \Theta_2 \qquad \Theta_2 \vdash e_2 \Leftarrow [\Theta_2]A_1 \dashv \Delta}{\Gamma \vdash e_1\ e_2 \Rightarrow A_2 \dashv \Delta}\ \textsc{AApp}$$

$$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$$

$$\frac{\Gamma, x : A \vdash e \Leftarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x.\ e \Leftarrow A \to B \dashv \Delta}\ \textsc{ALam} \qquad \frac{\Gamma, a \vdash e \Leftarrow A \dashv \Delta, a, \Theta}{\Gamma \vdash e \Leftarrow \forall a.A \dashv \Delta}\ \textsc{AGen}$$

$$\frac{\Gamma \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A \lesssim [\Theta]B \dashv \Delta}{\Gamma \vdash e \Leftarrow B \dashv \Delta}\ \textsc{ASub}$$

$$\boxed{\Gamma \vdash A \triangleright A_1 \to A_2 \dashv \Delta}$$

$$\frac{\Gamma, \widehat{a} \vdash A[a \mapsto \widehat{a}] \triangleright A_1 \to A_2 \dashv \Delta}{\Gamma \vdash \forall a.A \triangleright A_1 \to A_2 \dashv \Delta}\ \textsc{AM-Forall} \qquad \frac{}{\Gamma \vdash (A_1 \to A_2) \triangleright (A_1 \to A_2) \dashv \Gamma}\ \textsc{AM-Arr}$$

$$\frac{}{\Gamma \vdash \star \triangleright \star \to \star \dashv \Gamma}\ \textsc{AM-Unknown} \qquad \frac{}{\Gamma[\widehat{c}] \vdash \widehat{c} \triangleright \widehat{a} \to \widehat{b} \dashv \Gamma[\widehat{a}, \widehat{b}, \widehat{c} = \widehat{a} \to \widehat{b}]}\ \textsc{AM-Var}$$

Fig. 12. Algorithmic typing

them are quite standard. Perhaps rule AApp (which differs significantly from that in the DK system) deserves attention. It relies on the algorithmic matching judgment $\Gamma \vdash A \triangleright A_1 \to A_2 \dashv \Delta$. Rule AM-ForallL replaces a with a fresh existential variable $\widehat{a}$, thus eliminating guessing. Rule AM-Arr and AM-Unknown correspond directly to the declarative rules. Rule AM-Var, which has no corresponding declarative version, is similar to InstRArr/InstLArr: we create $\widehat{a}$ and $\widehat{b}$ and add $\widehat{c} = \widehat{a} \to \widehat{b}$ to the context.

5.3 Completeness and Soundness

We prove that the algorithmic rules are sound and complete with respect to the declarative specifications. We need an auxiliary judgment $\Gamma \longrightarrow \Delta$ that captures a notion of information increase from input contexts Γ to output contexts Δ [11].

Soundness. Roughly speaking, soundness of the algorithmic system says that given an expression e that type checks in the algorithmic system, there exists a corresponding expression e' that type checks in the declarative system. However there is one complication: e does not necessarily have more annotations than e'. For example, by ALAM we have $\lambda x.\ x \Leftarrow (\forall a.a) \to (\forall a.a)$, but $\lambda x.\ x$ itself cannot have type $(\forall a.a) \to (\forall a.a)$ in the declarative system. To circumvent that, we add an annotation to the lambda abstraction, resulting in $\lambda x : (\forall a.a).\ x$, which is typeable in the declarative system with the same type. To relate $\lambda x.\ x$ and $\lambda x : (\forall a.a).\ x$, we erase all annotations on both expressions. The definition of erasure $\lfloor \cdot \rfloor$ is standard and thus omitted.

Theorem 1 (Soundness of Algorithmic Typing). *Given $\Delta \longrightarrow \Omega$,*

1. *If $\Gamma \vdash e \Rightarrow A \dashv \Delta$ then $\exists e'$ such that $[\Omega]\Delta \vdash e' : [\Omega]A$ and $\lfloor e \rfloor = \lfloor e' \rfloor$.*
2. *If $\Gamma \vdash e \Leftarrow A \dashv \Delta$ then $\exists e'$ such that $[\Omega]\Delta \vdash e' : [\Omega]A$ and $\lfloor e \rfloor = \lfloor e' \rfloor$.*

Completeness. Completeness of the algorithmic system is the reverse of soundness: given a declarative judgment of the form $[\Omega]\Gamma \vdash [\Omega] \ldots$, we want to get an algorithmic derivation of $\Gamma \vdash \cdots \dashv \Delta$. It turns out that completeness is a bit trickier to state in that the algorithmic rules generate existential variables on the fly, so Δ could contain unsolved existential variables that are not found in Γ, nor in Ω. Therefore the completeness proof must produce another complete context Ω' that extends both the output context Δ, and the given complete context Ω. As with soundness, we need erasure to relate both expressions.

Theorem 2 (Completeness of Algorithmic Typing). *Given $\Gamma \longrightarrow \Omega$ and $\Gamma \vdash A$, if $[\Omega]\Gamma \vdash e : A$ then there exist Δ, Ω', A' and e' such that $\Delta \longrightarrow \Omega'$ and $\Omega \longrightarrow \Omega'$ and $\Gamma \vdash e' \Rightarrow A' \dashv \Delta$ and $A = [\Omega']A'$ and $\lfloor e \rfloor = \lfloor e' \rfloor$.*

6 Discussion

6.1 Top Types

To demonstrate that our definition of consistent subtyping (Definition 2) is applicable to other features, we show how to extend our approach to Top types with all the desired properties preserved.

In order to preserve the orthogonality between subtyping and consistency, we require $\top$ to be a common supertype of all static types, as shown in rule S-TOP. This rule might seem strange at first glance, since even if we remove the requirement *A static*, the rule seems reasonable. However, an important point is that because of the orthogonality between subtyping and consistency, subtyping itself should not contain a potential information loss! Therefore, subtyping instances such as $\star <: \top$ are not allowed. For consistency, we add the rule that $\top$ is consistent with $\top$, which is actually included in the original reflexive rule

$A \sim A$. For consistent subtyping, every type is a consistent subtype of $\top$, for example, $\mathsf{Int} \to \star \lesssim \top$.

$$\frac{A \ \mathit{static}}{\Psi \vdash A <: \top}\ \text{S-Top} \qquad \top \sim \top \qquad \frac{}{\Psi \vdash A \lesssim \top}\ \text{CS-Top}$$

It is easy to verify that Definition 2 is still equivalent to that in Fig. 7 extended with rule CS-Top. That is, Theorem 1 holds:

Proposition 4 (Extension with $\top$). *$\Psi \vdash A \lesssim B \Leftrightarrow \Psi \vdash A <: C$, $C \sim D$, $\Psi \vdash D <: B$, for some C, D.*

We extend the definition of concretization (Definition 3) with $\top$ by adding another equation $\gamma(\top) = \{\top\}$. Note that Castagna and Lanvin [7] also have this equation in their calculus. It is easy to verify that Proposition 2 still holds:

Proposition 5 (Equivalent to AGT on $\top$). *$A \lesssim B$ if only if $A \mathrel{\widetilde{<:}} B$.*

Siek and Taha's [22] *Definition of Consistent Subtyping Does Not Work for* $\top$. As the analysis in Sect. 3.2, $\mathsf{Int} \to \star \lesssim \top$ only holds when we first apply consistency, then subtyping. However we cannot find a type A such that $\mathsf{Int} \to \star <: A$ and $A \sim \top$. Also we have a similar problem in extending the restriction operator: *non-structural* masking between $\mathsf{Int} \to \star$ and $\top$ cannot be easily achieved.

6.2 Interpretation of the Dynamic Semantics

In Sect. 4.2 we have seen an example where a source expression could produce two different target expressions with different runtime behaviour. As we explained, this is due to the guessing nature of the declarative system, and from the typing point of view, no type is particularly better than others. However, in practice, this is not desirable. Let us revisit the same example, now from the algorithmic point of view (we omit the translation for space reasons):

$$f : \forall a.a \to a \vdash (\lambda x : \star.\ f\ x) \Rightarrow \star \to \widehat{a} \dashv f : \forall a.a \to a, \widehat{a}$$

Compared with declarative typing, which produces many types ($\star \to \mathsf{Int}$, $\star \to \mathsf{Bool}$, and so on), the algorithm computes the type $\star \to \widehat{a}$ with $\widehat{a}$ unsolved in the output context. What can we know from the output context? The only thing we know is that $\widehat{a}$ is not constrained at all! However, it is possible to make a more refined distinction between different kinds of existential variables. The first kind of existential variables are those that indeed have no constraints at all, as they do not affect the dynamic semantics. The second kind of existential variables (as in this example) are those where the only constraint is that *the variable was once compared with an unknown type* [12].

To emphasize the difference and have better support for dynamic semantics, we could have *gradual variables* in addition to existential variables, with the difference that only unsolved gradual variables are allowed to be unified with the unknown type. An irreversible transition from existential variables to gradual

variables occurs when an existential variable is compared with $\star$. After the algorithm terminates, we can set all unsolved existential variables to be any (static) type (or more precisely, as Garcia and Cimini [12], with *static type parameters*), and all unsolved gradual variables to be $\star$ (or *gradual type parameters*). However, this approach requires a more sophisticated declarative/algorithmic type system than the ones presented in this paper, where we only produce static monotypes in type inference. We believe this is a typical trade-off in existing gradual type systems with inference [12,23]. Here we suppress the complexity of dynamic semantics in favour of the conciseness of static typing.

6.3 The Dynamic Guarantee

In Sect. 4.3 we mentioned that the dynamic guarantee is closely related to the coherence issue. To aid discussion, we first give the definition of dynamic guarantee as follows:

Definition 5 (Dynamic guarantee). *Suppose* $e' \sqsubseteq e$, $\emptyset \vdash e : A \rightsquigarrow s$ *and* $\emptyset \vdash e' : A' \rightsquigarrow s'$, *if* $s \Downarrow v$, *then* $s' \Downarrow v'$ *and* $v' \sqsubseteq v$.

The dynamic guarantee says that if a gradually typed program evaluates to a value, then removing type annotations always produces a program that evaluates to an equivalent value (modulo type annotations). Now apparently the coherence issue of the declarative system breaks the dynamic guarantee. For instance:

$$(\lambda f : \forall a.a \to a.\ \lambda x : \mathsf{Int}.\ f\ x)\ (\lambda x.x)\ 3 \qquad (\lambda f : \forall a.a \to a.\ \lambda x : \star.\ f\ x)\ (\lambda x.x)\ 3$$

The left one evaluates to 3, whereas its less precise version (right) will give a cast error if a is instantiated to Bool for example.

As discussed in Sect. 6.2, we could design a more sophisticated declarative/algorithmic type system where coherence is retained. However, even with a coherent source language, the dynamic guarantee is still a question. Currently, the dynamic guarantee for our target language $\lambda\mathsf{B}$ is still an open question. According to Igarashi et al. [14], the difficulty lies in the definition of term precision that preserves the semantics.

7 Related Work

Along the way we discussed some of the most relevant work to motivate, compare and promote our gradual typing design. In what follows, we briefly discuss related work on gradual typing and polymorphism.

Gradual Typing. The seminal paper by Siek and Taha [21] is the first to propose gradual typing. The original proposal extends the simply typed lambda calculus by introducing the unknown type $\star$ and replacing type equality with type consistency. Later Siek and Taha [22] incorporated gradual typing into a

simple object oriented language, and showed that subtyping and consistency are orthogonal – an insight that partly inspired our work. We show that subtyping and consistency are orthogonal in a much richer type system with higher-rank polymorphism. Siek et al. [25] proposed a set of criteria that provides important guidelines for designers of gradually typed languages. Cimini and Siek [8] introduced the *Gradualizer*, a general methodology for generating gradual type systems from static type systems. Later they also develop an algorithm to generate dynamic semantics [9]. Garcia et al. [13] introduced the AGT approach based on abstract interpretation.

Gradual Type Systems with Explicit Polymorphism. Ahmed et al. [1] proposed $\lambda\mathsf{B}$ that extends the blame calculus [29] to incorporate polymorphism. The key novelty of their work is to use dynamic sealing to enforce parametricity. Devriese et al. [10] proved that embedding of System F terms into $\lambda\mathsf{B}$ is not fully abstract. Igarashi et al. [14] also studied integrating gradual typing with parametric polymorphism. They proposed System F_G, a gradually typed extension of System F, and System F_C, a new polymorphic blame calculus. As has been discussed extensively, their definition of type consistency does not apply to our setting (implicit polymorphism). All of these approaches mix consistency with subtyping to some extent, which we argue should be orthogonal.

Gradual Type Inference. Siek and Vachharajani [23] studied unification-based type inference for gradual typing, where they show why three straightforward approaches fail to meet their design goals. Their type system infers gradual types, which results in a complicated type system and inference algorithm. Garcia and Cimini [12] presented a new approach where gradual type inference only produces static types, which is adopted in our type system. They also deal with let-polymorphism (rank 1 types). However none of these works deals with higher-ranked implicit polymorphism.

Higher-Rank Implicit Polymorphism. Odersky and Läufer [17] introduced a type system for higher-rank types. Based on that, Peyton Jones et al. [18] developed an approach for type checking higher-rank predicative polymorphism. Dunfield and Krishnaswami [11] proposed a bidirectional account of higher-rank polymorphism, and an algorithm for implementing the declarative system, which serves as a sole inspiration for our algorithmic system. The key difference, however, is the integration of gradual typing. Vytiniotis et al. [28] defers static type errors to runtime, which is fundamentally different from gradual typing, where programmers can control over static or runtime checks by precision of the annotations.

8 Conclusion

In this paper, we present a generalized definition of consistent subtyping, which is proved to be applicable to both polymorphic and top types. Based on the new definition of consistent subtyping, we have developed a gradually typed calculus with predicative implicit higher-rank polymorphism, and an algorithm to implement it. As future work, we are interested to investigate if our results can scale to real world languages and other programming language features.

Acknowledgements. We thank Ronald Garcia and the anonymous reviewers for their helpful comments. This work has been sponsored by the Hong Kong Research Grant Council projects number 17210617 and 17258816.

References

1. Ahmed, A., Findler, R.B., Siek, J.G., Wadler, P.: Blame for all. In: Proceedings of the 38th Symposium on Principles of Programming Languages (2011)
2. Ahmed, A., Jamner, D., Siek, J.G., Wadler, P.: Theorems for free for free: parametricity, with and without types. In: Proceedings of the 22nd International Conference on Functional Programming (2017)
3. Schwerter, F.B., Garcia, R., Tanter, É.: A theory of gradual effect systems. In: Proceedings of the 19th International Conference on Functional Programming (2014)
4. Bierman, G., Meijer, E., Torgersen, M.: Adding dynamic types to C♯. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 76–100. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14107-2_5
5. Bierman, G., Abadi, M., Torgersen, M.: Understanding TypeScript. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 257–281. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44202-9_11
6. Bonnaire-Sergeant, A., Davies, R., Tobin-Hochstadt, S.: Practical optional types for clojure. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 68–94. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_4
7. Castagna, G., Lanvin, V.: Gradual typing with union and intersection types. Proc. ACM Program. Lang. **1**(ICFP), 41:1–41:28 (2017)
8. Cimini, M., Siek, J.G.: The gradualizer: a methodology and algorithm for generating gradual type systems. In: Proceedings of the 43rd Symposium on Principles of Programming Languages (2016)
9. Cimini, M., Siek, J.G.: Automatically generating the dynamic semantics of gradually typed languages. In: Proceedings of the 44th Symposium on Principles of Programming Languages (2017)
10. Devriese, D., Patrignani, M., Piessens, F.: Parametricity versus the universal type. Proc. ACM Program. Lang. **2**(POPL), 38 (2017)
11. Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. In: International Conference on Functional Programming (2013)
12. Garcia, R., Cimini, M.: Principal type schemes for gradual programs. In: Proceedings of the 42nd Symposium on Principles of Programming Languages (2015)
13. Garcia, R., Clark, A.M., Tanter, É.: Abstracting gradual typing. In: Proceedings of the 43rd Symposium on Principles of Programming Languages (2016)

14. Igarashi, Y., Sekiyama, T., Igarashi, A.: On polymorphic gradual typing. In: Proceedings of the 22nd International Conference on Functional Programming (2017)
15. Jafery, K.A., Dunfield, J.: Sums of uncertainty: refinements go gradual. In: Proceedings of the 44th Symposium on Principles of Programming Languages (2017)
16. Mitchell, J.C.: Polymorphic type inference and containment. In: Logical Foundations of Functional Programming (1990)
17. Odersky, M., Läufer, K.: Putting type annotations to work. In: Proceedings of the 23rd Symposium on Principles of Programming Languages (1996)
18. Jones, S.P., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. J. Funct. Program. **17**(1), 1–82 (2007)
19. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: Proceedings of the IFIP 9th World Computer Congress (1983)
20. Reynolds, J.C.: The coherence of languages with intersection types. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 675–700. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-54415-1_70
21. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Proceedings of the 2006 Scheme and Functional Programming Workshop (2006)
22. Siek, J., Taha, W.: Gradual typing for objects. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 2–27. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73589-2_2
23. Siek, J.G., Vachharajani, M.: Gradual typing with unification-based inference. In: Proceedings of the 2008 Symposium on Dynamic Languages (2008)
24. Siek, J.G., Wadler, P.: The key to blame: gradual typing meets cryptography (draft) (2016)
25. Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T.: Refined criteria for gradual typing. In: LIPIcs-Leibniz International Proceedings in Informatics (2015)
26. Verlaguet, J.: Facebook: analyzing PHP statically. In: Proceedings of Commercial Users of Functional Programming (2013)
27. Vitousek, M.M., Kent, A.M., Siek, J.G., Baker, J.: Design and evaluation of gradual typing for Python. In: Proceedings of the 10th Symposium on Dynamic Languages (2014)
28. Vytiniotis, D., Jones, S.P., Magalhães, J.P.: Equality proofs and deferred type errors: a compiler pearl. In: Proceedings of the 17th International Conference on Functional Programming, ICFP 2012, New York (2012)
29. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 1–16. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00590-9_1

HOBiT: Programming Lenses Without Using Lens Combinators

Kazutaka Matsuda[1(✉)] and Meng Wang[2]

[1] Tohoku University, Sendai 980-8579, Japan
kztk@ecei.tohoku.ac.jp
[2] University of Bristol, Bristol BS8 1TH, UK

Abstract. We propose HOBiT, a higher-order bidirectional programming language, in which users can write bidirectional programs in the familiar style of conventional functional programming, while enjoying the full expressiveness of lenses. A bidirectional transformation, or a lens, is a pair of mappings between source and view data objects, one in each direction. When the view is modified, the source is updated accordingly with respect to some laws—a pattern that is found in databases, model-driven development, compiler construction, and so on. The most common way of programming lenses is with lens combinators, which are lens-to-lens functions that compose simpler lenses to form more complex ones. Lens combinators preserve the bidirectionality of lenses and are expressive; but they compel programmers to a specialised point-free style—i.e., no naming of intermediate computation results—limiting the scalability of bidirectional programming. To address this issue, we propose a new bidirectional programming language HOBiT, in which lenses are represented as standard functions, and combinators are mapped to language constructs with binders. This design transforms bidirectional programming, enabling programmers to write bidirectional programs in a flexible functional style and at the same time access the full expressiveness of lenses. We formally define the syntax, type system, and the semantics of the language, and then show that programs in HOBiT satisfy bidirectionality. Additionally, we demonstrate HOBiT's programmability with examples.

1 Introduction

Transforming data from one format to another is a common task of programming: compilers transform program texts into syntax trees, manipulate the trees and then generate low-level code; database queries transform base relations into views; model transformations generate lower-level implementations from higher-level models; and so on. Very often, such transformations will benefit from being bidirectional, allowing changes to the targets to be mapped back to the sources too. For example, if one can run a compiler front-end (preprocessing, parsing, desugaring, etc.) backwards, then all sorts of program analysis tools will be able to focus on a much smaller core language, without sacrificing usability, as

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 31–59, 2018.
https://doi.org/10.1007/978-3-319-89884-1_2

their outputs in term of the core language will be transformed backwards to the source language. In the same way, such needs arise in databases (the *view-update problem* [1,6,12]) and model-driven engineering (bidirectional model transformation) [28,33,35].

As a response to this challenge, programming language researchers have started to design languages that execute deterministically in both directions, and the lens framework is the most prominent among all. In the lens framework, a *bidirectional transformation* (or a *lens*) $\ell \in \mathit{Lens}\ S\ V$, consists of $\mathit{get}\ \ell \in S \to V$, and $\mathit{put}\ \ell \in S \to V \to S$ [3,7,8]. (When clear from the context, or unimportant, we sometimes omit the lens name and write simply *get*/*put*.) Function *get* extracts a view from a source, and *put* takes both an updated view and the original source as inputs to produce an updated source. The additional parameter of *put* makes it possible to recover some of the source data that is not present in the view. In other words, *get* needs not to be injective to have a *put*. Not all pairs of *get*/*put* are considered correct lenses. The following round-triping laws of a lens ℓ are generally required to establish bidirectionality:

$$\mathit{put}\ \ell\ s\ v = s \quad \text{if} \quad \mathit{get}\ \ell\ s = v \qquad \textbf{(Acceptability)}$$
$$\mathit{get}\ \ell\ s' = v \quad \text{if} \quad \mathit{put}\ \ell\ s\ v = s' \qquad \textbf{(Consistency)}$$

for all s, s' and v. (In this paper we write $e = e'$ with the assumption that neither e nor e' is undefined. Stronger variants of the laws enforcing totality exist elsewhere, for example in [7].) Here *consistency* ensures that all updates on a view are captured by the updated source, and *acceptability* prohibits changes to the source if no update has been made on the view. Collectively, the two laws defines *well-behavedness* [1,7,12].

The most common way of programming lenses is with lens combinators [3,7,8], which are basically a selection of lens-to-lens functions that compose simpler lenses to form more complex ones. This combinator-based approach follows the long history of lightweight language development in functional programming. The distinctive advantage of this approach is that by restricting the lens language to a few selected combinators, well-behavedness can be more easily preserved in programming, and therefore given well-behaved lenses as inputs, the combinators are guaranteed to produce well-behaved lenses. This idea of lens combinators is very influential academically, and various designs and implementations have been proposed [2,3,7–9,16,17,27,32] over the years.

1.1 The Challenge of Programmability

The complexity of a piece of software can be classified as either intrinsic or accidental. Intrinsic complexity reflects the inherent difficulty of the problem at hand, whereas accidental complexity arises from the particular programming language, design or tools used to implement the solution. This work aims at reducing the accidental complexity of bidirectional programming by contributing to the design of bidirectional languages. In particularly, we identify a language restriction—i.e., no naming of intermediate computation results—which complicates lens programming, and propose a new design that removes it.

As a teaser to demonstrate the problem, let us consider the list append function. In standard unidirectional programming, it can be defined simply as $append\ x\ y = \mathbf{case}\ x\ \mathbf{of}\ \{[\,] \to y;\ a : x' \to a : append\ x'\ y\}$. Astute readers may have already noticed that *append* is defined by structural recursion on x, which can be made explicit by using *foldr* as in $append\ x\ y = foldr\ (:)\ y\ x$.

But in a lens language based on combinators, things are more difficult. Specifically, *append* now requires a more complicated recursion pattern, as below.

```
appendL :: Lens ([A], [A]) [A]
appendL =
  cond idL (λ_.True) (λ_.λ_.[]) (consL ◦̂ (idL × appendL)) (not ∘ null) (λ_.λ_.⊥)
  ◦̂ rearr ◦̂ (outListL × idL)
 where outListL :: Lens [A] (Either ( ) (A, [A]))
       rearr    :: Lens (Either ( ) (a, b), c) (Either c (a, (b, c)))
       (◦̂)      :: Lens b c → Lens a b → Lens a c
       cond     :: Lens a c → ... → Lens b c → ... → Lens (Either a b) c
       ...
```

It is beyond the scope of this paper to explain how exactly the definition of *appendL* works, as its obscurity is what this work aims to remove. Instead, we informally describe its behaviour and the various components of the code. The above code defines a lens: forwards, it behaves as the standard *append*, and backwards, it splits the updated view list, and when the length of the list changes, this definition implements (with the grayed part) the bias of keeping the length of the first source list whenever possible (to disambiguate multiple candidate source changes). Here, *cond*, $(\hat{\circ})$, etc. are lens combinators and *outListL* and *rearr* are auxiliary lenses, as can be seen from their types. Unlike its unidirectional counterpart, *appendL* can no longer be defined as a structural recursion on list; instead it traverses a pair of lists with rather complex rearrangement *rearr*.

Intuitively, the additional grayed parts is intrinsic complexity, as they are needed for directing backwards execution. However, the complicated recursion scheme, which is a direct result of the underlying limitation of lens languages, is certainly accidental. Recall that in the definition of *append*, we were able to use the variable y, which is bound outside of the recursion pattern, inside the body of *foldr*. But the same is not possible with lens combinators which are strictly 'pointfree'. Moreover, even if one could name such variables (points), their usage with lens combinators will be very restricted in order to guarantee well-behavedness [21,23]. This problem is specific to opaque non-function objects such as lenses, and goes well beyond the traditional issues associated with the pointfree programming style.

In this paper, we design a new bidirectional language HOBiT, which aims to remove much of the accidental difficulty found in combinator-based lens programming, and reduces the gap between bidirectional programming and standard functional programming. For example, the following definition in HOBiT implements the same lens as *appendL*.

$$
\begin{aligned}
&\mathit{appendB} :: \mathbf{B}[A] \to \mathbf{B}[A] \to \mathbf{B}[A] \\
&\mathit{appendB}\ x\ y = \underline{\mathbf{case}}\ x\ \underline{\mathbf{of}}\ [\,] \quad \to y \qquad\qquad\qquad\qquad \underline{\mathbf{with}}\ \lambda\_.\mathsf{True}\ \ \underline{\mathbf{by}}\ (\lambda\_.\lambda\_.[\,]) \\
&\qquad\qquad\qquad\qquad\quad a : x' \to a \mathbin{\underline{:}} \mathit{appendB}\ x'\ y\ \underline{\mathbf{with}}\ \mathit{not} \circ \mathit{null}\ \underline{\mathbf{by}}\ (\lambda\_.\lambda\_.\bot)
\end{aligned}
$$

As expected, the above code shares the grayed part with the definition of *appendL* as the two implement the same backwards behaviour. The difference is that *appendB* uses structural recursion in the same way as the standard unidirectional *append*, greatly simplifying programming. This is made possible by the HOBiT's type system and semantics, allowing unrestricted use of free variables. This difference in approach is also reflected in the types: *appendB* is a proper function (instead of the abstract lens type of *appendL*), which readily lends itself to conventional functional programming. At the same time, *appendB* is also a proper lens, which when executed by the HOBiT interpreter behave exactly like *appendL*. A major technical challenge in the design of HOBiT is to guarantee this duality, so that functions like *appendB* are well-behaved by construction despite the flexibility in their construction.

1.2 Contributions

As we can already see from the very simple example above, the use of HOBiT simplifies bidirectional programming by removing much of the accidental complexity. Specifically, HOBiT stands out from existing bidirectional languages in two ways:

1. It supports the conventional programming style that is used in unidirectional programming. As a result, a program in HOBiT can be defined in a way similar to how one would define only its *get* component. For example, *appendB* is defined in the same way as the unidirectional *append*.
2. It supports incremental improvement. Given the very often close resemblance of a bidirectional-program definition and that of its *get* component, it becomes possible to write an initial version of a bidirectional program almost identical to its *get* component and then to adjust the backwards behaviour gradually, without having to significantly restructure the existing definition.

Thanks to these distinctive advantages, HOBiT for the first time allows us to construct realistically-sized bidirectional programs with relative ease. Of course, this does not mean free lunch: the ability to control backwards behaviours will not magically come without additional code (for example the grayed part above). What HOBiT achieves is that programming effort may now focus on the productive part of specifying backwards behaviours, instead of being consumed by circumventing language restrictions.

In summary, we make the following contributions in this paper.

– We design a higher-order bidirectional programming language HOBiT, which supports convenient bidirectional programming with control of backwards behaviours (Sect. 3). We also discuss several extensions to the language (Sect. 5).

– We present the semantics of HOBiT inspired by the idea of staging [5], and prove the well-behavedness property using Kripke logical relations [18] (Sect. 4).
– We demonstrate the programmability of HOBiT with examples such as desugaring/resugaring [26] (Sect. 6). Additional examples including a bidirectional evaluator for λ-calculus [21,23], a parser/printer for S-expressions, and bookmark extraction for Netscape [7] can be found at https://bitbucket.org/kztk/hibx together with a prototype implementation of HOBiT.

2 Overview: Bidirectional Programming Without Combinators

In this section, we informally introduce the essential constructs of HOBiT and demonstrate their use by a few small examples. Recall that, as seen in the *appendB* example, the strength of HOBiT lies in allowing programmers to access λ-abstractions without restrictions on the use of λ-bound variables.

2.1 The case Construct

The most important language construct in HOBiT is **case** (pronounced as *bidirectional case*), which provides pattern matching and easy access to bidirectional branching, and also importantly, allows unrestricted use of λ-bound variables.

In general, a **case** expression has the following form.

$$\textbf{case}\ e\ \textbf{of}\ \{p_1 \to e_1\ \textbf{with}\ \phi_1\ \textbf{by}\ \rho_1; \ldots; p_n \to e_n\ \textbf{with}\ \phi_n\ \textbf{by}\ \rho_n\}$$

(Like Haskell, we shall omit "{", "}" and ";" if they are clear from the layout.) In the type system of HOBiT, a **case**-expression has type $\mathbf{B}B$, if e and e_i have types $\mathbf{B}A$ and $\mathbf{B}B$, and ϕ_i and ρ_i have types $B \to \mathit{Bool}$ and $A \to B \to A$, where A and B contains neither $(\to)$ nor $\mathbf{B}$. The type $\mathbf{B}A$ can be understood intuitively as "updatable A". Typically, the source and view data are given such $\mathbf{B}$-types, and a function of type $\mathbf{B}A \to \mathbf{B}B$ is the HOBiT equivalent of *Lens A B*.

The pattern matching part of **case** performs two implicit operations: it first unwraps the $\mathbf{B}$-typed value, exposing its content for normal pattern matching, and then it wraps the variables bound by the pattern matching, turning them into 'updatable' $\mathbf{B}$-typed values to be used in the bodies. For example, in the second branch of *appendB*, a and x' can be seen as having types A and $[A]$ in the pattern, but $\mathbf{B}A$ and $\mathbf{B}[A]$ types in the body; and the bidirectional constructor $(\underline{:}) :: \mathbf{B}A \to \mathbf{B}[A] \to \mathbf{B}[A]$ combines them to produce a $\mathbf{B}$-typed list.

In addition to the standard conditional branches, **case**-expression has two unique components ϕ_i and ρ_i called *exit conditions* and *reconciliation functions* respectively, which are used in backwards executions. Exit condition ϕ_i is an over-approximation of the forwards-execution results of the expressions e_i. In other words, if branch i is choosen, then $\phi_i\ e_i$ must evaluate to True. This assertion is checked dynamically in HOBiT, though could be checked statically with

a sophisticated type system [7]. In the backwards direction the exit condition is used for deciding branching: the branch with its exit condition satisfied by the updated view (when more than one match, the original branch used in the forwards direction has higher priority) will be picked for execution. The idea is that due to the update in the view, the branch taken in the backwards direction may be different from the one taken in the original forwards execution, a feature that is commonly supported by lens languages [7] which we call *branch switching*.

Branch switching is crucial to *put*'s *robustness*, i.e., the ability to handle a wide range of view updates (including those affect the branching decisions) without failing. We explain its working in details in the following.

Branch Switching. Being able to choose a different branch in the backwards direction only solves part of the problem. Let us consider the case where a forward execution chooses the n^{th} branch, and the backwards execution, based on the updated view, chooses the m^{th} ($m \neq n$) branch. In this case, the original value of the pattern-matched expression e, which is the reason for the n^{th} branch being chosen, is not compatible with the *put* of the m^{th} branch.

As an example, let us consider a simple function that pattern-matches on an *Either* structure and returns an list. Note that we have purposely omitted the reconciliation functions.

```
f :: B(Either [A] (A, [A])) → B[A]
f x = case x of Left ys       → ys      with λ_.True  {- no by here -}
                Right (y, ys) → y : ys  with not ∘ null
```

We have said that functions of type $\mathbf{B}A \to \mathbf{B}B$ are also fully functioning lenses of type *Lens A B*. In HOBiT, the above code runs as follows, where `HOBiT>` is the prompt of HOBiT's read-eval-print loop, and `:get` and `:put` are meta-language operations to perform *get* and *put* respectively.

```
HOBiT> :get f (Left [1, 2, 3])
[1, 2, 3]
HOBiT> :get f (Right (1, [2, 3]))
[1, 2, 3]
HOBiT> :put f (Left [1, 2, 3]) [4, 5]     -- The view [1, 2, 3] is updated to [4, 5].
Left [4, 5]                               -- Both exit conditions are true with [4, 5],
                                          -- so the original branch (Left) is taken.
HOBiT> :put f (Right (1, [2, 3])) [4, 5]
Right (4, [5])                            -- Similar, but the original branch is Right.
HOBiT> :put f (Right (1, [2, 3])) []
⊥                                         -- Branch switches, but computation fails.
```

As we have explained above, exit conditions are used to decide which branch will be used in the backwards direction. For the first and second evaluations of *put*, the exit conditions corresponding to the original branches were true for the updated view. For the last evaluation of *put*, since the exit condition of

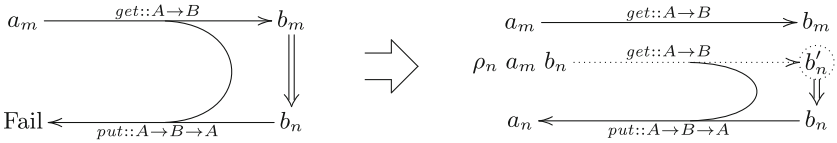

Fig. 1. Reconciliation function: assuming exit conditions ϕ_m and ϕ_n where $\phi_m\ b_n =$ False but $\phi_n\ b_n =$ True, and reconciliation functions ρ_m and ρ_n.

the original branch was false but that of the other branch was true, branch switching is required here. However, a direct *put*-execution of f with the inputs (Right $(1, [2, 3])$) and $[\,]$ crashes (represented by $\bot$ above), for a good reason, as the two inputs are in an inconsistent state with respect to f.

This is where reconciliation functions come into the picture. For the Left branch above, a sensible reconciliation function will be $(\lambda\_.\lambda\_.\mathsf{Left}\ [\,])$, which when applied turns the conflicting source (Right $(1, [2, 3])$) into Left $[\,]$, and consequently the *put*-execution may succeed with the new inputs and returns Left $[\,]$. It is not difficult to verify that the "reconciled" *put*-execution still satisfies well-behavedness. Note that despite the similarity in types, reconciliation functions are not *put*; they merely provide a default source value to allow stuck *put*-executions to proceed. We visualise the effect of reconciliation functions in Fig. 1. The left-hand side is bidirectional execution without successful branch-switching, and since $\phi_m\ b_n$ is false (indicating that b_n is not in the range of the m^{th} branch) the execution of *put* must (rightfully) fail in order to guarantee well-behavedness. On the right-hand side, reconciliation function ρ_n produces a suitable source from a_m and b_n (where $\phi_n\ (get\ (\rho_n\ a_m\ b_n))$ is True), and *put* executes with b_n and the new source $\rho_n\ a_m\ b_n$. It is worth mentioning that branch switching with reconciliation functions does not compromise correctness: though the quality of the user-defined reconciliation functions affects robustness as they may or may not be able to resolve conflicts, successful *put*-executions always guarantee well-behavedness, regardless the involvement of reconciliation functions.

Revisiting *appendB*. Recall *appendB* from Sect. 1.1 (reproduced below).

$$
\begin{array}{l}
appendB :: \mathbf{B}[A] \to \mathbf{B}[A] \to \mathbf{B}[A] \\
appendB\ x\ y = \underline{\mathbf{case}}\ x\ \underline{\mathbf{of}}\ [\,] \quad \to y \qquad\qquad\qquad\qquad \underline{\mathbf{with}}\ \lambda\_.\mathsf{True} \quad \underline{\mathbf{by}}\ (\lambda\_.\lambda\_.[\,]) \\
\qquad\qquad\qquad\qquad\quad\ a : x' \to a : appendB\ x'\ y\ \underline{\mathbf{with}}\ not \circ null\ \underline{\mathbf{by}}\ (\lambda\_.\lambda\_.\bot)
\end{array}
$$

The exit condition for the nil case always returns true as there is no restriction on the value of y, and for the cons case it requires the returned list to be non-empty. In the backwards direction, when the updated view is non-empty, both exit conditions will be true, and then the original branch will be taken. This means that since *appendB* is defined as a recursion on x, the backwards execution will try to unroll the original recursion step by step (i.e., the cons branch will be taken for a number of times that is the same as the length of x) as long as the view remains non-empty. If an updated view list is shorter than x, then $not \circ null$

will become false before the unrolling finishes, and the nil branch will be taken (branch-switching) and the reconciliation function will be called.

The definition of *appendB* is curried; straightforward uncurrying turns it into the standard form $\mathbf{B}A \to \mathbf{B}B$ that can be interpreted by HOBiT as a lens. The following HOBiT program is the bidirectional variant of *uncurry*.

$$\begin{aligned}&\mathit{uncurryB} :: (\mathbf{B}A \to \mathbf{B}B \to \mathbf{B}C) \to \mathbf{B}(A, B) \to \mathbf{B}C\\ &\mathit{uncurryB}\ f\ z = \underline{\mathbf{let}}\ (x, y) = z\ \underline{\mathbf{in}}\ f\ x\ y\end{aligned}$$

Here, $\underline{\mathbf{let}}\ p = e\ \underline{\mathbf{in}}\ e'$ is syntactic sugar for $\underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \{p \to e'\ \underline{\mathbf{with}}\ (\lambda\_.\mathsf{True})\ \underline{\mathbf{by}}\ (\lambda s.\lambda\_.s)\}$, in which the reconciliation function is never called as there is only one branch. Let $\mathit{appendB}' = \mathit{uncurryB}\ \mathit{appendB}$, then we can run $\mathit{appendB}'$ as:

```
HOBiT> :get appendB' ([1,2],[3,4,5])
[1,2,3,4,5]
HOBiT> :put appendB' ([1,2],[3,4,5]) [6,7,8,9,10]
([6,7],[8,9,10])      -- No structural change, no branch switching.
HOBiT> :put appendB' ([1,2],[3,4,5]) [6,7]
([6,7],[])      -- No branch switching, still.
HOBiT> :put appendB' ([1,2],[3,4,5]) [6]
([6],[])      -- Branch-switching happens and the recursion terminates early.
```

Difference from Lens Combinators. As mentioned above, the idea of branch switching can be traced back to lens languages. In particular, the design of **case** is inspired by the combinator *cond* [7]. Despite the similarities, it is important to recognise that **case** is not only a more convenient syntax for *cond*, but also crucially supports the unrestricted use of λ-bound variables. This more fundamental difference is the reason why we could define *appendB* in the conventional functional style as the variables x and y are used freely in the body of **case**. In other words, the novelty of HOBiT is its ability to combine the traditional (higher-order) functional programming and the bidirectional constructs as found in lens combinators, effectively establishing a new way of bidirectional programming.

2.2 A More Elaborate Example: *linesB*

In addition to supporting convenient programming and robustness in *put* execution, the **case** constructs can also be used to express intricate details of backwards behaviours. Let us consider the *lines* function in Haskell as an example, which splits a string into a list of strings by newlines, for example, *lines* `"AA\nBB\n"` = [`"AA"`, `"BB"`], except that the last newline character in its input is optional. For example, *lines* returns [`"AA"`, `"BB"`] for both `"AA\nBB\n"` and `"AA\nBB"`. Suppose that we want the backwards transformation of *lines* to exhibit a behaviour that depends on the original source:

```
linesB :: BString → B[String]
linesB str =
  let (f, b) = breakNLB str
  in case b of '\n' : x : r → f ⋮ linesB (x ⋮ r)
                                  with (> 1) ∘ length by (λb.λ_.'\n' : ' ' : b)
               b'           → f ⋮ [] with (== 1) ∘ length by (λb.λ_.lastNL b)
  where {lastNL [] = []; lastNL ['\n'] = ['\n']; lastNL (a : x) = lastNL x}
breakNLB :: BString → B(String, String)
breakNLB str = case str of
  []      → ([], [])                          with p1 by (λ_.λ_.[])
  '\n' : s → ([], '\n' ⋮ s)                   with p2 by (λ_.λ_."\n")
  c : s   → let (f, r) = breakNLB s in (c ⋮ f, r) with p3 by (λ_.λ_." ")
  where {p1(x, y) = null y; p2(x, y) = null x && not (null y); p3(x, y) = not (null x)}
```

Fig. 2. *linesB* and *breakNLB*

```
HOBiT> :put linesB "AA\nBB" ["a", "b"]
"a\nb"
HOBiT> :put linesB "AA\nBB" ["a", "b", "c"]
"a\nb\nc"
HOBiT> :put linesB "AA\nBB" ["a"]
"a"
HOBiT> :put linesB "AA\nBB\n" ["a", "b", "c"]
"a\nb\nc\n"
HOBiT> :put linesB "AA\nBB\n" ["a"]
"a\n"
```

This behaviour is achieved by the definition in Fig. 2, which makes good use of reconciliation functions. Note that we do not consider the contrived corner case where the string ends with duplicated newlines such as in `"A\n\n"`. The function *breakNLB* splits a string at the first newline; since *breakNLB* is injective, its exit conditions and reconciliation functions are of little interest. The interesting part is in the definition of *linesB*, particularly its use of reconciliation functions to track the existence of a last newline character. We firstly explain the branching structure of the program. On the top level, when the first line is removed from the input, the remaining string b may contain more lines, or be the end (represented by either the empty list or the singleton list `['\n']`). If the first branch is taken, the returned result will be a list of more than one element. In the second branch when it is the end of the text, b could contain a newline or simply be empty. We do not explicitly give patterns for the two cases as they have the same body $f \mathbin{⋮} []$, but the reconciliation function distinguishes the two in order to preserve the original source structure in the backwards execution. Note that we intentionally use the same variable name b in the case analysis and the reconciliation function, to signify that the two represent the same source data. The use of argument b in the reconciliation functions serves the purpose of remembering the (non)existence of the last newline in the original source, which is then preserved in the new source.

$$
\begin{aligned}
e &::= x \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{True} \mid \mathsf{False} \mid [\,] \mid e_1 : e_2 \mid \mathbf{case}\ e\ \mathbf{of}\ \{p_i \to e_i\}_{i=1,2} \mid \mathbf{fix}\ (\lambda f.e) \\
&\quad \mid \underline{\mathsf{True}} \mid \underline{\mathsf{False}} \mid \underline{[\,]} \mid e_1 \mathbin{\underline{:}} e_2 \mid \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ e_i'\ \underline{\mathbf{by}}\ e_i''\}_{i=1,2} \\
p &::= x \mid \mathsf{True} \mid \mathsf{False} \mid [\,] \mid p_1 : p_2
\end{aligned}
$$

Fig. 3. Syntax of HOBiT Core

It is worth noting that just like the other examples we have seen, this definition in HOBiT shares a similar structure with a definition of *lines* in Haskell.[1] The notable difference is that a Haskell definition is likely to have a different grouping of the three cases of *lines* into two branches, as there is no need to keep track of the last newline for backwards execution. Recall that reconciliation functions are called *after* branches are chosen by exit conditions; in the case of *linesB*, the reconciliation function is used to decide the reconciled value of b' to be `"\n"` or `""`. This, however, means that we cannot separate the pattern b' into two `"\n"` and `""` with copying its branch body and exit condition, because then we lose a chance to choose a reconciled value of b based on its original value.

3 Syntax and Type System of HOBiT Core

In this section, we describe the syntax and the type system of the core of HOBiT.

3.1 Syntax

The syntax of HOBiT Core is given in Fig. 3. For simplicity, we only consider booleans and lists. The syntax is almost the same as the standard λ-calculus with the fixed-point combinator (**fix**), lists and booleans. For data constructors and case expressions, there are in addition bidirectional versions that are underlined. We allow the body of **fix** to be non-λs to make our semantics simple (Sect. 4), though such a definition like $\mathbf{fix}(\lambda x.\mathsf{True} : x)$ can diverge.

Although in examples we used **case**/**case**-expressions with an arbitrary number of branches having overlapping patterns under the first-match principle, we assume for simplicity that in HOBiT Core **case**/**case**-expressions must have exactly two branches whose patterns do not overlap; extensions to support these features are straightforward. As in Haskell, we sometimes omit the braces and semicolons if they are clear from the layout.

[1] Haskell's *lines*'s behaviour is a bit more complicated as it returns [] if and only if the input is `""`. This behaviour can be achieved by calling *linesB* only when the input list is nonempty.

$$\boxed{\Gamma;\Delta \vdash e : A}$$

$$\frac{\Gamma(x) = A}{\Gamma;\Delta \vdash x : A} \quad \frac{\Delta(x) = \sigma}{\Gamma;\Delta \vdash x : \mathbf{B}\sigma} \quad \frac{\Gamma, x : A;\Delta \vdash e : B}{\Gamma;\Delta \vdash \lambda x.e : A \to B} \quad \frac{\Gamma;\Delta \vdash e_1 : A \to B \quad \Gamma;\Delta \vdash e_2 : A}{\Gamma;\Delta \vdash e_1\ e_2 : B}$$

$$\frac{\Gamma, f : A;\Delta \vdash e : A}{\Gamma;\Delta \vdash \mathbf{fix}(\lambda f.e) : A} \quad \frac{}{\Gamma;\Delta \vdash \mathsf{True} : \mathit{Bool}} \quad \frac{}{\Gamma;\Delta \vdash \mathsf{False} : \mathit{Bool}} \quad \frac{}{\Gamma;\Delta \vdash [\,] : [A]}$$

$$\frac{\Gamma;\Delta \vdash e_1 : A \quad \Gamma;\Delta \vdash e_2 : [A]}{\Gamma;\Delta \vdash e_1 : e_2 : [A]} \quad \frac{}{\Gamma;\Delta \vdash \underline{\mathsf{True}} : \mathbf{B}\mathit{Bool}} \quad \frac{}{\Gamma;\Delta \vdash \underline{\mathsf{False}} : \mathbf{B}\mathit{Bool}} \quad \frac{}{\Gamma;\Delta \vdash \underline{[\,]} : \mathbf{B}[\sigma]}$$

$$\frac{\Gamma;\Delta \vdash e_1 : \mathbf{B}\sigma \quad \Gamma;\Delta \vdash e_2 : \mathbf{B}[\sigma]}{\Gamma;\Delta \vdash e_1 \mathbin{\underline{:}} e_2 : \mathbf{B}[\sigma]} \quad \frac{\Gamma;\Delta \vdash e : A \quad \Gamma_i \vdash p_i : A \quad \Gamma, \Gamma_i;\Delta \vdash e_i : B \quad (i = 1, 2)}{\Gamma;\Delta \vdash \mathbf{case}\ e\ \mathbf{of}\ \{p_i \to e_i\}_{i=1,2} : B}$$

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash e : \mathbf{B}\sigma \quad \Delta_i \vdash p_i : \sigma \quad \Gamma;\Delta, \Delta_i \vdash e_i : \mathbf{B}\tau \\ \Gamma;\Delta \vdash e'_i : \tau \to \mathit{Bool} \quad \Gamma;\Delta \vdash e''_i : \sigma \to \tau \to \sigma \quad (i = 1, 2)\end{array}}{\Gamma;\Delta \vdash \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ e'_i\ \underline{\mathbf{by}}\ e''_i\}_{i=1,2} : \mathbf{B}\tau}$$

$$\boxed{\Gamma \vdash p : A}$$

$$\frac{}{x : A \vdash x : A} \quad \frac{}{\emptyset \vdash \mathsf{True} : \mathit{Bool}} \quad \frac{}{\emptyset \vdash \mathsf{False} : \mathit{Bool}} \quad \frac{}{\emptyset \vdash [\,] : [A]} \quad \frac{\Gamma_1 \vdash e_1 : A \quad \Gamma_2 \vdash e_2 : [A]}{\Gamma_1, \Gamma_2 \vdash e_1 : e_2 : [A]}$$

Fig. 4. Typing rules: $\Delta \vdash p : \sigma$ is similar to $\Gamma \vdash p : A$ but asserts that the resulting environment is actually a bidirectional environment.

3.2 Type System

The types in HOBiT Core are defined as follows.

$$A, B ::= \mathbf{B}\sigma \mid A \to B \mid [A] \mid \mathit{Bool}$$

We use the metavariable $\sigma, \tau, \ldots$ for types that do not contain $\to$ nor $\mathbf{B}$, We call σ-types *pure datatypes*, which are used for sources and views of lenses. Intuitively, $\mathbf{B}\sigma$ represents "updatable σ"—data subject to update in bidirectional transformation. We keep the type system of HOBiT Core simple, though it is possible to include polymorphic types or intersection types to unify unidirectional and bidirectional constructors.

The typing judgment $\Gamma;\Delta \vdash e : A$, which reads that under environments Γ and Δ, expression e has type A, is defined by the typing rules in Fig. 4. We use two environments: Δ (the *bidirectional type environment*) is for variables introduced by pattern-matching through **case**, and Γ for everything else. It is interesting to observe that Δ only holds pure datatypes, as the pattern variables of **case** have pure datatypes, while Γ holds any types. We assume that the variables in Γ and those in Δ are disjoint, and appropriate α-renaming has been done to ensure this. This separation of Δ from Γ does not affect typeability, but is key to our semantics and correctness proof (Sect. 4). Most of the rules are standard except **case**; recall that we only use unidirectional constructors in patterns which have pure types, while the variables bound in the patterns are used as $\mathbf{B}$-typed values in branch bodies.

4 Semantics of HOBiT Core

Recall that the unique strength of HOBiT is its ability to mix higher-order unidirectional programming with bidirectional programming. A consequence of this mixture is that we can no longer specify its semantics in the same way as other *first-order* bidirectional languages such as [13], where two semantics—one for *get* and the other for *put*—suffice. This is because the category of lenses is believed to have no exponential objects [27] (and thus does not permit λs).

4.1 Basic Idea: Staging

Our solution to this problem is staging [5], which separates evaluation into two stages: the unidirectional parts is evaluated first to make way for a bidirectional semantics, which only has to deal with the residual first-order programs. As a simple example, consider the expression $(\lambda z.z)\ (x \mathbin{\underline{:}} ((\lambda w.w)\ y) \mathbin{\underline{:}} \underline{[]})$. The first-stage evaluation, $e \Downarrow_{\mathrm{U}} E$, eliminates λs from the expression as in $(\lambda z.z)\ (x \mathbin{\underline{:}} ((\lambda w.w)\ y) \mathbin{\underline{:}} \underline{[]}) \Downarrow_{\mathrm{U}} x \mathbin{\underline{:}} y \mathbin{\underline{:}} \underline{[]}$. Then, our bidirectional semantics will be able to treat the residual expression as a lens between value environments and values, following [13,20]. Specifically, we have the *get* evaluation relation $\mu \vdash_{\mathrm{G}} E \Rightarrow v$, which computes the value v of E under environment μ as usual, and the *put* evaluation relation $\mu \vdash_{\mathrm{P}} v \Leftarrow E \dashv \mu'$, which computes an updated environment μ' for E from the updated view v and the original environment μ. In pseudo syntax, it can be understood as $put\ E\ \mu\ v = \mu'$, where μ represents the original source and μ' the new source.

It is worth mentioning that a complete separation of the stages is not possible due to the combination of **fix** and **case**, as an attempt to fully evaluate them in the first stage will result in divergence. Thus, we delay the unidirectional evaluation inside **case** to allow **fix**, and consequently the three evaluation relations (uni-directional, *get*, and *put*) are mutually dependent.

4.2 Three Evaluation Relations: Unidirectional, *get* and *put*

First, we formally define the set of residual expressions:

$$\begin{aligned} E ::= {} & \mathsf{True} \mid \mathsf{False} \mid [\,] \mid E_1 : E_2 \mid \lambda x.e \\ \mid {} & x \mid \underline{\mathsf{True}} \mid \underline{\mathsf{False}} \mid \underline{[\,]} \mid E_1 \mathbin{\underline{:}} E_2 \mid \underline{\mathbf{case}}\ E_0\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ E_i\ \underline{\mathbf{by}}\ E_i'\}_{i=1,2} \end{aligned}$$

They are treated as values in the unidirectional evaluation, and as expressions in the *get* and *put* evaluations. Notice that e or e_i appear under λ or **case**, meaning that their evaluations are delayed.

The set of *(first-order) values* is defined as below.

$$v ::= \mathsf{True} \mid \mathsf{False} \mid [\,] \mid v_1 : v_2$$

Accordingly, we define a *(first-order) value environment* μ as a finite mapping from variables to first-order values.

$$\frac{}{x \Downarrow_{\mathrm{U}} x} \qquad \frac{e_1 \Downarrow_{\mathrm{U}} \lambda x.e \quad e_2 \Downarrow_{\mathrm{U}} E_2 \quad e[E_2/x] \Downarrow_{\mathrm{U}} E}{e_1\ e_2 \Downarrow_{\mathrm{U}} E} \qquad \frac{}{\lambda x.e \Downarrow_{\mathrm{U}} \lambda x.e} \qquad \frac{e[\mathbf{fix}(\lambda f.e)/f] \Downarrow_{\mathrm{U}} E}{\mathbf{fix}(\lambda f.e) \Downarrow_{\mathrm{U}} E}$$

$$\frac{e_0 \Downarrow_{\mathrm{U}} E_0 \quad e_i' \Downarrow_{\mathrm{U}} E_i' \quad e_i'' \Downarrow_{\mathrm{U}} E_i'' \quad (i = 1, 2)}{\underline{\mathbf{case}}\ e_0\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ e_i'\ \underline{\mathbf{by}}\ e_i''\}_{i=1,2} \Downarrow_{\mathrm{U}} \underline{\mathbf{case}}\ E_0\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ E_i'\ \underline{\mathbf{by}}\ E_i''\}_{i=1,2}}$$

Fig. 5. Evaluation rules for unidirectional parts (excerpt)

Unidirectional Evaluation Relation. The rules for the unidirectional evaluation relation is rather standard, as excerpted in Fig. 5. The bidirectional constructs (i.e., bidirectional constructors and **case**) are frozen, i.e., behave just like ordinary constructors in this evaluation. Notice that we can evaluate an expression containing free variables; then the resulting residual expression may contain the free variables.

Bidirectional (*get* and *put*) Evaluation Relations. The *get* and *put* evaluation relations, $\mu \vdash_{\mathrm{G}} E \Rightarrow v$ and $\mu \vdash_{\mathrm{P}} v \Leftarrow E \dashv \mu'$, are defined so that they together form a lens.

Weakening of Environment. Before we lay out the semantics, it is worth explaining a subtlety in environment handling. In conventional evaluation semantics, a larger than necessary environment does no harm, as long as there is no name clashes. For example, whether the expression x is evaluated under the environment $\{x = 1\}$ or $\{x = 1, y = 2\}$ does not matter. However, the same is not true for bidirectional evaluation. Let us consider a residual expression $E = x \mathbin{\underline{:}} y \mathbin{\underline{:}} \underline{[\,]}$, and a value environment $\mu = \{x = 1, y = 2\}$ as the original source. We expect to have $\mu \vdash_{\mathrm{G}} E \Rightarrow 1 : 2 : [\,]$, which may be derived as:

$$\frac{\dfrac{}{\mu \vdash_{\mathrm{G}} x \Rightarrow 1} \quad \dfrac{\vdots}{\mu \vdash_{\mathrm{G}} y \mathbin{\underline{:}} \underline{[\,]} \Rightarrow 2 : [\,]}}{\mu \vdash_{\mathrm{G}} x \mathbin{\underline{:}} y \mathbin{\underline{:}} \underline{[\,]} \Rightarrow 1 : 2 : [\,]}$$

In the *put* direction, for an updated view say $3 : 4 : [\,]$, we expect to have $\mu \vdash_{\mathrm{P}} 3 : 4 : [\,] \Leftarrow E \dashv \{x = 3, y = 4\}$ with the corresponding derivation:

$$\frac{\dfrac{}{\mu \vdash_{\mathrm{P}} 3 \Leftarrow x \dashv ?_1} \quad \dfrac{\vdots}{\mu \vdash_{\mathrm{P}} 4 : [\,] \Leftarrow y \mathbin{\underline{:}} \underline{[\,]} \dashv ?_2}}{\mu \vdash_{\mathrm{P}} 3 : 4 : [\,] \Leftarrow x \mathbin{\underline{:}} y \mathbin{\underline{:}} \underline{[\,]} \dashv \{x = 3, y = 4\}}$$

What shall the environments $?_1$ and $?_2$ be? One way is to have $\mu \vdash_{\mathrm{P}} 3 \Leftarrow x \dashv \{x = 3, y = 2\}$, and $\mu \vdash_{\mathrm{P}} 4 : [\,] \Leftarrow y \mathbin{\underline{:}} \underline{[\,]} \dashv \{x = 1, y = 4\}$, where the variables do not appear free in the residual expression takes their values from the original source environment μ. However, the evaluation will get stuck here, as there is no reasonable way to produce the expected result $\{x = 3, y = 4\}$ from $?_1 = \{x = 3, y = 2\}$ and $?_2 = \{x = 1, y = 4\}$. In other words, the redundancy in environment is harmful as it may cause conflicts downstream.

Our solution to this problem, which follows from [21–23,29], is to allow *put* to return value environments containing only bindings that are relevant for the residual expressions under evaluation. For example, we have $\mu \vdash_{\mathrm{P}} 3 \Leftarrow x \dashv \{x = 3\}$, and $\mu \vdash_{\mathrm{P}} 4 : [\,] \Leftarrow y \mathbin{\underline{:}} [\,] \dashv \{y = 4\}$. Then, we can merge the two value environments $?_1 = \{x = 3\}$ and $?_2 = \{y = 4\}$ to obtain the expected result $\{x = 3, y = 4\}$. As a remark, this seemingly simple solution actually has a non-trivial effect on the reasoning of well-behavedness. We defer a detailed discussion on this to Sect. 4.3.

Now we are ready to define *get* and *put* evaluation rules for each bidirectional constructs. For variables, we just lookup or update environments. Recall that μ is a mapping (i.e., function) from variables to (first-order) values, while we use a record-like notation such as $\{x = v\}$.

$$\frac{}{\mu \vdash_{\mathrm{G}} x \Rightarrow \mu(x)} \qquad \frac{}{\mu \vdash_{\mathrm{P}} v \Leftarrow x \dashv \{x = v\}}$$

For constants $\underline{c}$ where $c = \mathsf{False}, \mathsf{True}, [\,]$, the evaluation rules are straightforward.

$$\frac{}{\mu \vdash_{\mathrm{G}} \underline{c} \Rightarrow c} \qquad \frac{}{\mu \vdash_{\mathrm{P}} c \Leftarrow \underline{c} \dashv \emptyset}$$

The above-mentioned behaviour of the bidirectional cons expression $E_1 \mathbin{\underline{:}} E_2$ is formally given as:

$$\frac{\mu \vdash_{\mathrm{G}} E_1 \Rightarrow v_1 \quad \mu \vdash_{\mathrm{G}} E_2 \Rightarrow v_2}{\mu \vdash_{\mathrm{G}} E_1 \mathbin{\underline{:}} E_2 \Rightarrow v_1 : v_2} \qquad \frac{\mu \vdash_{\mathrm{P}} v_1 \Leftarrow E_1 \dashv \mu_1' \quad \mu \vdash_{\mathrm{P}} v_2 \Leftarrow E_2 \dashv \mu_2'}{\mu \vdash_{\mathrm{P}} v_1 : v_2 \Leftarrow E_1 \mathbin{\underline{:}} E_2 \dashv \mu_1' \curlyvee \mu_2'}$$

(Note that the variable rules guarantee that only free variables in the residual expressions end up in the resulting environments.) Here, $\curlyvee$ is the merging operator defined as: $\mu \curlyvee \mu' = \mu \cup \mu'$ if there is no x such that $\mu(x) \neq \mu'(x)$. For example, $\{x = 3\} \curlyvee \{y = 4\} = \{x = 3, y = 4\}$, and $\{x = 3, y = 4\} \curlyvee \{y = 4\} = \{x = 3, y = 4\}$, but $\{x = 3, y = 2\} \curlyvee \{y = 4\}$ is undefined.

The most interesting rules are for **case**. In the *get* direction, it is not different from the ordinary **case** except that exit conditions are asserted, as shown in Fig. 6. We use the following predicate for pattern matching.

$$\mathit{match}(p_k, v_0, \mu_k) \;=\; (p_k\mu_k = v_0) \wedge (\mathsf{dom}(\mu_k) = \mathsf{fv}(p_k))$$

Here, we abuse the notation to write $p_k\mu_k$ for the value obtained from p_k by replacing the free variables x in p_k with $\mu_k(x)$. One might notice that we have the disjoint union $\mu \uplus \mu_i$ in Fig. 6 where μ_i holds the values of the variables in p_i, as we assume α-renaming of bound variables that is consistent in *get* and *put*. Recall that p_1 and p_2 are assumed not to overlap, and hence the evaluation is deterministic. Note that the reconciliation functions E_i'' are untouched by the rule.

The *put* evaluation rule of **case** shown in Fig. 6 is more involved. In addition to checking which branch should be chosen by using exit conditions, we need two rules to handle the cases with and without branch switching. Basically,

$$\frac{\mu \vdash_{\mathrm{G}} E_0 \Rightarrow v_0 \quad match(p_i, v_0, \mu_i) \quad e_i \Downarrow_{\mathrm{U}} E_i \quad \mu \uplus \mu_i \vdash_{\mathrm{G}} E_i \Rightarrow v \quad E_i' \; v \Downarrow_{\mathrm{U}} \mathsf{True}}{\mu \vdash_{\mathrm{G}} \underline{\mathbf{case}} \; E_0 \; \mathbf{of} \; \left\{p_i \to e_i \; \underline{\mathbf{with}} \; E_i' \; \underline{\mathbf{by}} \; E_i''\right\}_{i=1,2} \Rightarrow v}$$

$$\frac{\begin{array}{c}\mu \vdash_{\mathrm{G}} E_0 \Rightarrow v_0 \quad match(p_i, v_0, \mu_i) \quad E_i' \; v \Downarrow_{\mathrm{U}} \mathsf{True} \quad e_i \Downarrow_{\mathrm{U}} E_i \\ \mu \uplus \mu_i \vdash_{\mathrm{P}} v \Leftarrow E_i \dashv \mu' \uplus_{\mathsf{dom}(\mu),\mathsf{dom}(\mu_i)} \mu_i' \quad v_0' = p_i(\mu_i' \triangleleft \mu_i) \quad \mu \vdash_{\mathrm{P}} v_0' \Leftarrow E_0 \dashv \mu_0'\end{array}}{\mu \vdash_{\mathrm{P}} v \Leftarrow \underline{\mathbf{case}} \; E_0 \; \mathbf{of} \; \left\{p_i \to e_i \; \underline{\mathbf{with}} \; E_i' \; \underline{\mathbf{by}} \; E_i''\right\}_{i=1,2} \dashv \mu_0' \curlyvee \mu'}$$

$$\frac{\begin{array}{c}\mu \vdash_{\mathrm{G}} E_0 \Rightarrow v_0 \quad match(p_i, v_0, \mu_i) \quad E_i' \; v \Downarrow_{\mathrm{U}} \mathsf{False} \quad j = 3 - i \quad E_j' \; v \Downarrow_{\mathrm{U}} \mathsf{True} \quad e_j \Downarrow_{\mathrm{U}} E_j \\ E_j'' \; v_0 \; v \Downarrow_{\mathrm{U}} u_0 \quad match(p_j, u_0, \mu_j) \\ \mu \uplus \mu_j \vdash_{\mathrm{P}} v \Leftarrow E_j \dashv \mu' \uplus_{\mathsf{dom}(\mu),\mathsf{dom}(\mu_j)} \mu_j' \quad v_0' = p_j(\mu_j' \triangleleft \mu_j) \quad \mu \vdash_{\mathrm{P}} v_0' \Leftarrow E_0 \dashv \mu_0'\end{array}}{\mu \vdash_{\mathrm{P}} v \Leftarrow \underline{\mathbf{case}} \; E_0 \; \mathbf{of} \; \left\{p_i \to e_i \; \underline{\mathbf{with}} \; E_i' \; \underline{\mathbf{by}} \; E_i''\right\}_{i=1,2} \dashv \mu_0' \curlyvee \mu'}$$

Fig. 6. *get*- and *put*-Evaluation of **case**: we write $\mu \uplus_{X,Y} \mu'$ to ensure that $\mathsf{dom}(\mu) \subseteq X$ and $\mathsf{dom}(\mu') \subseteq Y$.

the branch to be taken in the backwards direction is decided first, by the *get*-evaluation of the case condition E_0 and the checking of the exit condition E_i' against the updated view v. After that, the body of the chosen branch e_i is firstly uni-directionally evaluated, and then its residual expression E_i is *put*-evaluated. The last step is *put*-evaluation of the case-condition E_0. When branch switching happens, there is the additional step of applying the reconciliation function E_j''.

Note the use of operator $\triangleleft$ in computing the updated case condition v_0'.

$$(\mu' \triangleleft \mu)(x) = \begin{cases} \mu'(x) & \text{if } x \in \mathsf{dom}(\mu') \\ \mu(x) & \text{otherwise} \end{cases}$$

Recall that in the beginning of this subsection, we discussed our approach of avoiding conflicts by producing environments with only relevant variables. This means the μ_i' above contains only variables that appear free in E_i, which may or may not be all the variables in p_i. Since this is the point where these variables are introduced, we need to supplement μ_i' with μ_i from the original pattern matching so that p_i can be properly instantiated.

Construction of Lens. Let us write $\mathcal{L}_0[\![E]\!]$ for a lens between value environments and values, defined as:

$$\begin{array}{ll} get \; \mathcal{L}_0[\![E]\!] \; \mu = v & \text{if } \mu \vdash_{\mathrm{G}} E \Rightarrow v \\ put \; \mathcal{L}_0[\![E]\!] \; \mu \; v = \mu' & \text{if } \mu \vdash_{\mathrm{P}} v \Leftarrow E \dashv \mu' \end{array}$$

Then, we can define the lens $\mathcal{L}[\![e]\!]$ induced from e (a closed function expression), where $e \; x \Downarrow_{\mathrm{U}} E$ for some fresh variable x.

$$\begin{array}{l} get \; \mathcal{L}[\![e]\!] \; s = get \; \mathcal{L}_0[\![E]\!] \; \{x = s\} \\ put \; \mathcal{L}[\![e]\!] \; s \; v = (\mu' \triangleleft \{x = s\})(x) \quad \text{where } \mu' = put \; \mathcal{L}_0[\![E]\!] \; \{x = s\} \; v \end{array}$$

Actually, `:get` and `:put` in Sect. 2 are realised by *get* $\mathcal{L}[\![e]\!]$ and *put* $\mathcal{L}[\![e]\!]$.